An Approach to Multiclass Classification with Fully Quantum Convolutional Neural Networks (QCNNs) and Analysis of Gradient and Gradient-Free Optimization - Third Draft

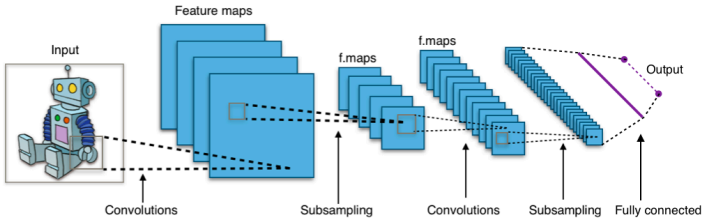
Himank Handa

This paper explores the implementation and optimization of fully quantum convolutional neural networks (QCNNs) for multi-class image classification. Building upon existing QCNN architectures, we investigate the viability of these models for complex classification tasks using the MNIST handwritten digits dataset. A key focus is the analysis of both gradient-based (ADAM) and gradient-free (COBYLA) optimization methods for training QCNNs. We examine the impact of various factors, including data set size, loss function, and circuit modifications, on the performance of binary and multi-class (4-class) classification. Our results demonstrate the potential of QCNNs for achieving comparable accuracy to classical CNNs, particularly with gradient-based optimization, while also highlighting the challenges associated with gradient-free optimization in higher dimensional Hilbert spaces. Furthermore, we observe a potential advantage of QCNNs in mitigating overfitting compared to classical counterparts. We also investigate the impact of hybrid quantum-classical architectures, where a classical linear layer is incorporated after the quantum circuit. Finally, we discuss future research directions, including dense encoding, optimization algorithm development, architectural changes, and noise analysis, to further enhance the efficiency and accuracy of QCNNs. All code and testing details are available on Github.

1. A Brief Introduction to Convolutional Neural Networks:

In the actively developing field of artificial intelligence, strategies for image processing and analysis have been found to be very effective for various applications, ranging from simple classification tasks to use in autonomous vehicles. Among these innovations, Convolutional Neural Networks (CNNs) have remained a foundational method to classify images, enabling computers to understand the complex patterns behind image data, interpreting them with exceptional accuracy and efficiency. This paper discusses an implementation of CNNs on Quantum Hardware using Quantum Convolutional Neural Networks (QCNNs) as well as adaptations to the architecture to support multi-class image classification.

CNNs are a class of deep neural networks designed to handle data, such as images, by employing a series of convolutional and pooling layers. CNNs are designed to recognize and extract distinct features like curves and straight lines from input images by employing convolutional layers. These convolutional layers use filters, also known as kernels, that are arrays, usually smaller than the input image, parameterized during training. The application of the filter over the entirety of the input image results in a feature map (detecting certain characteristics of the image), also known as an activation map. Subsequently, the pooling layers work to reduce the dimensions of the feature maps without losing essential information. This structure of convolution and pooling layers allows CNNs to understand and recognize patterns within the input data. Finally, a fully connected layer is used to make predictions based on the interpreted data and give the desired output. These operations help facilitate the neural network's ability to classify images with a great deal of precision.



(1a) [Aphex34, Wikimedia Commons]

CNNs have proven to be extremely effective in the field of artificial intelligence. However, as algorithms have become more complicated and data sets become much larger, classical computing has shown to not be able to facilitate these more complex applications. Consequently, Quantum Computing has emerged as a possible solution to the ever-growing complexity of AI. Quantum

Convolutional Neural Networks (QCNNs) are a recent innovation that combines the power of Quantum Computing with the proven capabilities of CNNs. The implemented architecture takes advantage of fewer parameters as well as inheriting the benefits of using the Hilbert space for quantum computing and manipulating multiple qubits in parallel [2]. However, there is not much debate, at the current stage, about the superiority of classical machine learning over quantum machine learning for classical processing tasks.

Although the advantages of QCNNs on Classical Data over classical computing are unclear, their usefulness in classifying Quantum Data is not. QCNNs, as discussed by the original authors of the paper, are intended to find intricate patterns in Quantum Data that may not be extractable through purely classical machine learning. In particular, the QCNN architecture in the paper was shown to be an efficient quantum classifier, leveraging its connection to the Multiscale Entanglement Renormalization Ansatz to identify whether an input quantum state resided within a specific phase of matter. It demonstrated, through the accurate recognition of a one-dimensional symmetry-protected topological phase, a significantly improved sample complexity compared to traditional methods based on string order parameters. Furthermore, QCNNs were applied to the optimization of quantum error correction codes, where the circuit's structure was exploited to optimize encoding and decoding schemes, outperforming existing codes like the Shor code under certain conditions [1].

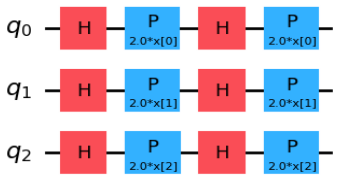
This paper will build upon the latest work in Quantum Convolutional Neural Networks, diving deep into the technical configurations of models. Specifically, we will look at circuit performance using gradient-free optimizers and examine the general viability of multi-class classification using Quantum Neural Networks, proposing new circuit architectures. Since gradient calculations for quantum circuits generally scale up linearly (in the best case) or exponentially (in the worst), they can be extremely inefficient to calculate, bringing up whether gradient-free optimizers might offer a better experience for much deeper circuit possibilities in the future. We discuss our findings for their viability in complex and multi-class scenarios.

2. Quantum Parallelism to Classical CNNs:

The QCNN structure used is similar to conventional CNNs: a combination of convolutional layers paired with pooling layers. The technical aspects, however, are different.

2.1 Input Data Translation:

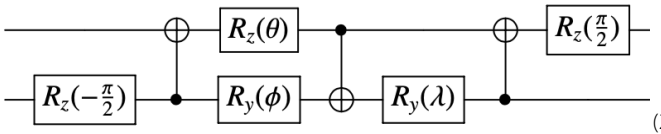
QCNNs need the input data to be encoded into a quantum state ρ_{in} in order to process it through a quantum circuit [1]. This paper utilizes ZFeatureMap (different from feature maps in classical CNNs), an existing function within the Qiskit Circuit Library [2], which takes the input parameters and converts them into a quantum circuit. This results in a 1:1 translation of pixels to qubits. This circuit is used to convert image data to quantum circuits later in the paper. The following shows a diagram of a 3-qubit circuit:



(2a)

2.2 The Convolutional Layer:

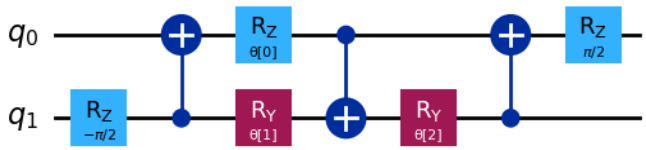
The convolutional layer is applied using a series of two-qubit unitary operators. Any two-qubit parameterized unitary would work, as its variables could be modified through training [2]. In this paper, we utilize the following 4x4 unitary matrix decomposition: $U = (A_1 \otimes A_2) \cdot N(\alpha, \beta, \gamma) \cdot (A_3 \otimes A_4)$. By using only $N(\alpha, \beta, \gamma)$, [3] limits the parameters to $\alpha, \beta, and\ \gamma$ to decrease training time. Here, the single-qubit rotations (A_1, A_2, A_3, A_4) are ignored for the time being. The relationship between the parameters being trained by the model and $\alpha, \beta, and\ \gamma$ is expressed as the following: $\alpha = \frac{\pi}{2} - 2\theta, \beta = 2\phi - \frac{\pi}{2}, \gamma = \frac{\pi}{2} - 2\lambda$. When simplified, the circuit is represented as $N(\alpha, \beta, \gamma) = exp(i[\alpha\sigma_x\sigma_x + \beta\sigma_y\sigma_y + \gamma\sigma_z\sigma_z])$:



(2b)

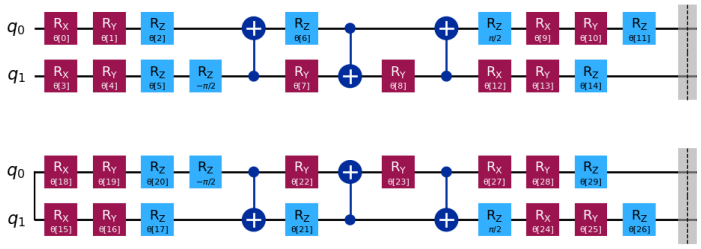
The following defines a function to implement the convolutional circuit between two qubits, acting as the equivalent of a "kernel" in a classical CNN to extract certain features.

```
def conv_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)
    target.cx(1, 0)
    target.rz(np.pi / 2, 0)
    return target
```



(2c)

The convolutional layer is made up of these two-qubit unitaries applied across neighboring qubits (2-qubit convolutional layer):

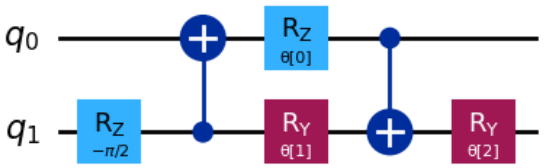


(2d)

2.3 The Pooling Layer:

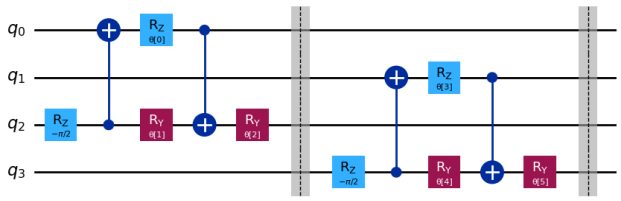
The pooling circuit will encode the information of one qubit onto another, letting us disregard the first. Every pooling layer will reduce the dimensionality by two. This effectively limits the number of parameters to only log(N) parameters for a circuit of size N qubits. This can be useful for the efficiency of the algorithm as well as computing gradients for quantum architectures. For binary classification, we will measure the last qubit that has encoded information from all other qubits to determine the output. [Corresponding to the observable $\sum_{n=1}^{i} I_n + Z_{i'}$ where i is the index of the last qubit]. We use the unitary presented in [3]:

```
def conv_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)
    target.cx(1, 0)
    target.rz(np.pi / 2, 0)
    return target
```



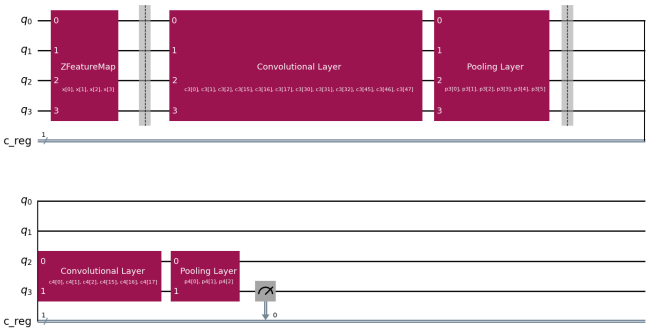
(2e)

The pooling layer is applied to pairs of qubits such as in the following four to two-dimensionality reduction:



(2f)

Finally, these are combined into a final circuit, with alternating convolutional and pooling layers (below is the diagram of a 4-qubit circuit as an example):



(2g) ***The measurement in the picture is only representational. In practice, we applied a Z-Basis Measurement through the observable.*** This implementation bypasses the flattening step (fully connected layer), leaving one qubit to measure with a value of -1 or 1. The measurement value of the qubit corresponds to one of the two classes. EstimatorQNN will run this circuit over for a certain number of shots to return the expectation value.

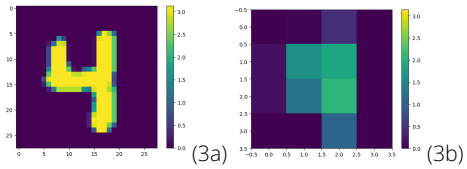
From this point, the algorithm functions in the same manner as a classical neural network. Using the final value(s), the loss is computed using a specified loss function, which is MSE-Loss for most of our binary tests and Cross-Entropy in multi-class cases (or when one-hot encoding is tested). Then, the optimizer, in this case, COBYLA, is able to train the weights of the parameterized convolutional and pooling layers (for the gradient-based optimizer, it also needs to calculate the quantum gradient). We explore how to expand this to multi-class classification using the MNIST numbers dataset after a short discussion of the binary implementation's pitfalls and techniques that will translate to our later testing.

3. Binary Implementation Methodology

This paper builds a modified version of the QCNN using the base developed in Section 2 in order to implement a (Multiclass; Section 4) handwritten digit images classifier, documenting numerous novel changes to improve upon the originally proposed model. Unlike other methods that introduce classical layers through a fully connected layer or create a hybrid layer, which can often prevent us from understanding the true efficacy of quantum layers, this paper will focus on **fully quantum** implementations *(no classical parameters to mask performance)*.

3.1 Input Data and Circuit Architecture:

We import the MNIST Handwritten Digits dataset from Keras datasets for training and testing of our model. Pixel values are numbered from 0 to 255, and, as in standard machine learning practice, we scale these values to range from 0 to pi (instead of [0,1] to correspond with rotation). Images in this dataset have a dimensionality of 28x28, containing 784 pixels in total. As discussed in *2.1*, this would translate to 784 qubits. This would not be possible to run on current quantum computers, let alone be simulated on a classical computer. Therefore, we reduce the dimensionality to 4x4 pixels, giving us 16 qubits to work with. This data is linearized to a single 1x16 array that goes into a 16-qubit ZFeatureMap[*2.1*].



(3a) (3b)

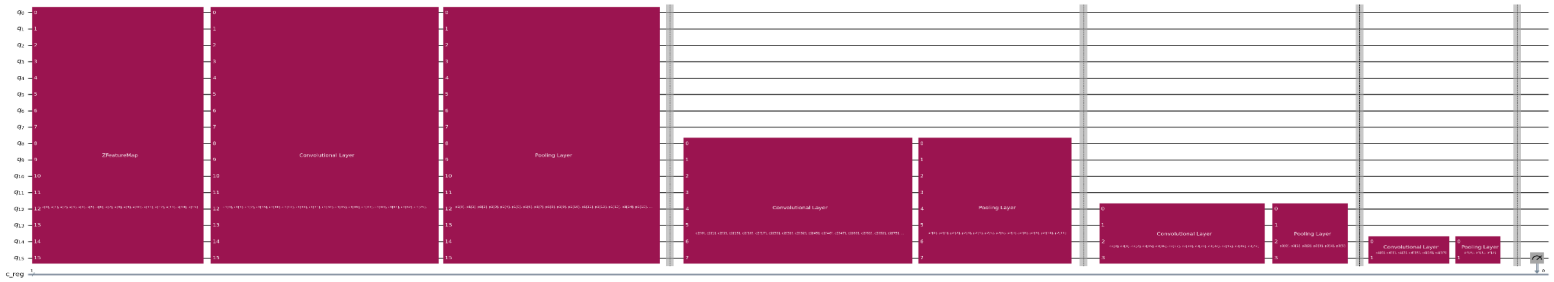
Figure 3a shows the initial scaled image and Figure 3b shows the resized image (using tf "area" resize for least quality loss and anti-aliasing). This example helps highlight the data loss due to this method.

3.2 Representation of Full Circuit & Ansatz:

The initial binary testing uses the architecture in [2] as shown below. After applying the feature map, we repeat convolutional and pooling layers until we have information about the image entirely encoded within the last qubit. Measuring this qubit in the $\boldsymbol{Z}$ Basis yields -1 or +1, corresponding to the two classes that can be predicted by the classifier.

3.3 Training:

We utilize Qiskit's EstimatorQNN Class as well as the Torch Connector Module as the base of our Neural

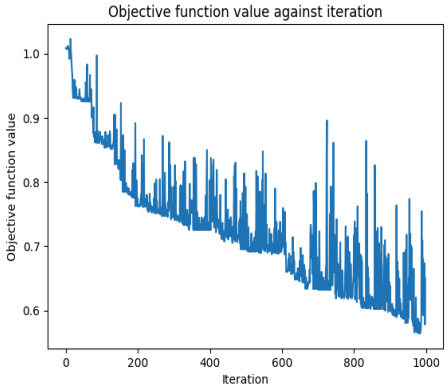


Network, which will allow us to integrate training within PyTorch for some of our tests. The EstimatorQNN Class uses an underlying Estimator Primitive that runs the circuit for a certain number of shots, returning an expectation value. This will allow us to calculate loss effectively.

For the first test, however, we used Qiskit Machine Learning's prebuilt NeuralNetworkClassifier module to set a baseline.

3.4 Baseline Results (Summary at 3.6):

The results indicated considerably high performance at all three checkpoints for both the Testing and Training Datasets.



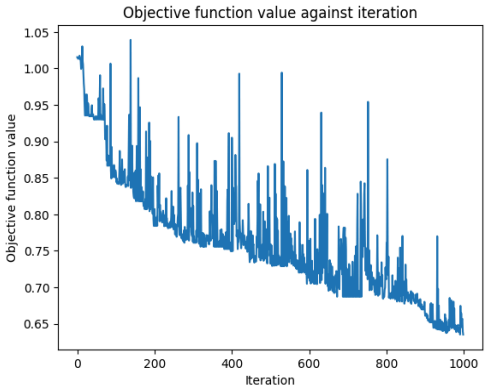
Accuracy Measurements at 333, 666, and 1000th epoch:
Train: [81.36%, 83.47%, 87.29%]
Test: [81.25%, 80.08%, 86.72%]
Config: [Workflow: QiskitML, Optimizer: COBYLA (Gradient-Free), Loss - MSE, Classes: 2, Iters: 1000, Batch-Size: 256, Rest Args: Default]
(3d)

(3c)

3.5 Training Data Sample Analysis for Gradient-Free Optimizer:

COBYLA is a gradient-free optimizer, which is why increasing the data set size only increased the execution time linearly. Our findings showed that the train data size seemed to have little effect on the convergence of the model. In fact, tests with a 128-batch size (half of the baseline configuration), while simultaneously halving the execution time, also occasionally yielded higher accuracies for both training and testing data, not displaying the common problem that gradient-based optimizers have with overfitting on smaller datasets. As long as the data size was more than 100, we observed no difference in the results of the model (upper 80% accuracy).

128 Images Train Data Set Result:



Config: [Workflow: QiskitML, Optimizer: COBYLA (Gradient-Free), Loss - MSE, Classes: 2, Iters: 1000, Batch-Size: 128, Rest Args: Default]
Accuracy Measurements at 333, 666, and 1000th epoch:
Train: [86.44%, 88.14%, 89.41%]
Test: [81.25%, 85.55%, 89.45%]
(3e)

3.6 Loss Function Analysis:

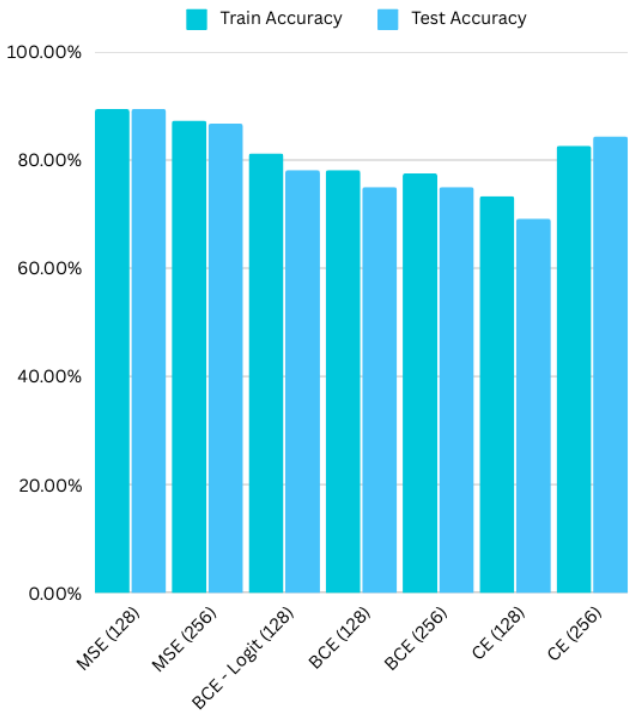
We also benchmarked numerous loss methods for later use in our multi-class, more complex circuits as well as the foundational changes to the architecture to support them. Our findings are presented here.

We experimented with Mean Squared Loss, Binary Cross Entropy Loss, as well as Cross Entropy Loss for the Binary Classification setup. For BCE Logits and MSE, we did not change the architecture. However, for BCE, we added one Sigmoid layer to transform the results to the interval [0,1]. To implement CE Loss, we added two observables on the last two qubits and used one-hot encoding to calculate loss by applying a Softmax layer at the end of the PyTorch module.

MSE Loss consistently performed better than both BCE and CE loss. Ultimately, however, there was only a marginal and inconsistent performance difference between the different losses for this gradient-free optimizer, indicating indifference to loss functions as long as they correlated closely with accuracy.

Quick Summary Table 2-Class Loss Functions

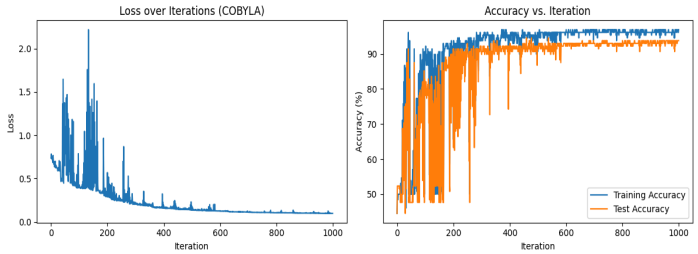
Loss Type	Train Accuracy	Test Accuracy
MSE (128)	89.41%	89.45%
MSE (256)	87.29%	86.72%
BCE - Logit (128)	81.25%	78.13%
BCE (128)	78.13%	75.00%
BCE (256)	77.54%	75.00%
CE (128)	77.12%	72.66%
CE (256)	82.63%	84.38%



3f: Results are best of multiple trials.

3.7 Classical Comparison:

To benchmark these results against a well-performing classical setup, we ran some tests on a simple CNN model using Pytorch with fair data and a 128-train images data set. The results using gradient-free optimizers showed faster convergence as well as a marginally higher accuracy compared to all quantum counterparts overall.



(3f)

Config: [Workflow: PyTorch, Optimizer: COBYLA (Gradient-Free), Loss - CrossEntropyLoss, Classes: 2, Iters: 1000, Batch-Size: 128, Rest Args: Default] Training Accuracy Measurements at 333, 666, and 1000th epoch:
Train: [92.97%, 96.09%, 96.88%]; Test (Final Only): [93.75%]

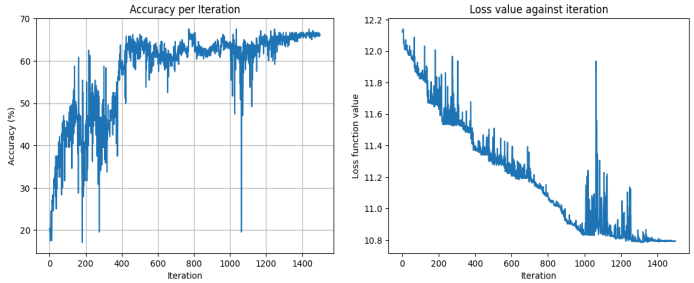
4. Facilitating Multiclass Classification:

We will be adopting our architecture to allow for 4-class prediction as proof of the efficacy of a purely quantum QCNN architecture to extend to multiclass. To facilitate Multiclass Classification, a couple of changes were made to the underlying architecture. Since the current architecture currently terminates with just one qubit, it is not possible to predict more than two classes. Indeed, it would require at least 2 qubits to predict 4 classes.

We use 4 qubits at the end of the architecture, removing the final pooling layer and utilizing one-hot encoding, where the highest sampling probability corresponds to the class predicted. Through some limited multiclass testing, this method was found to be more expressive, producing better results overall, while other implementations involving fewer qubits indicated problems with skewing toward certain classes. More testing should be done to see the relationship between the number of measured qubits and the expressibility of the circuit.

Finally, the measurement in the $\boldsymbol{Z}$ Basis is extended from the final qubit to the final 4 qubits. The expectation values are processed through a softmax layer that transforms them into probability scores. Although this architecture might be limited to predicting classes up to a fewer number of classes than there are qubits in the feature map, future work can be performed to understand whether extending the model to interact (through

entanglement) with additional qubits not part of the initial feature map could help alleviate this problem. The following shows the accuracy and loss trend:

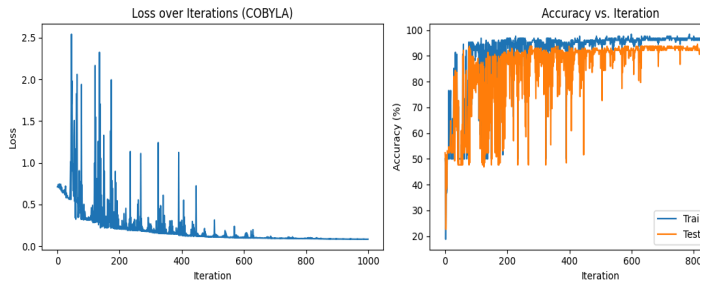


(4a)

Config: [Workflow: PyTorch, Optimizer: COBYLA (Gradient-Free), Loss - CrossEntropyLoss, Classes: 4, Iters: 1500, Batch-Size: 128, Rest Args: Default]
Training Accuracy Measurements at 500, 1000, and 15000th epoch:
Train: [65.0%, 63.75%, 66.25%]; Test (Final Only): [51.17%]

Although the training accuracy approached 70%, the testing accuracy seemed to trail significantly behind, always falling just above 50%. Furthermore, the flattening of the loss curve indicates that the optimizer may be getting stuck in local minima. Indeed, numerous trials indicated the vulnerability of gradient-free algorithms to fall into such local minima with similar levels of overfitting.

To benchmark this result against a classical set-up, we use a similar model as used in Figure [3g] but it sorts 4 classes.



(4b)

Config: [Workflow: PyTorch, Optimizer: COBYLA (Gradient-Free), Loss - CrossEntropyLoss, Classes: 4, Iters: 1000, Batch-Size: 128, Rest Args: Default]
Training Accuracy Measurements at 333, 666, and 1000th epoch:
Train: [96.88%, 96.09%, 96.88%]; Test (Final Only): [92.97%]

The significantly higher accuracy in the classical setting indicates either that the function being optimized in the quantum setting is highly non-convex (with several local minima) or that the gradient-free algorithm is unable to adequately traverse the Hilbert Space.

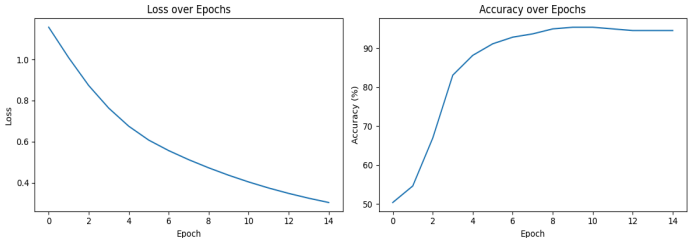
5. Gradient-Based Optimizer

Alongside gradient-free optimizers, we also test a gradient-based optimizer (ADAM) for our quantum circuits to better understand the difference in their performance compared to COBYLA.

We use ReverseEstimatorGradient for our gradient calculation method, which was found to be effective for our classical simulations. Out of the models tested with different learning rates, the configurations with a learning rate of 0.1 seemed to deliver the best results for our scenario. [See Github]

5.1 Binary Classification using ADAM:

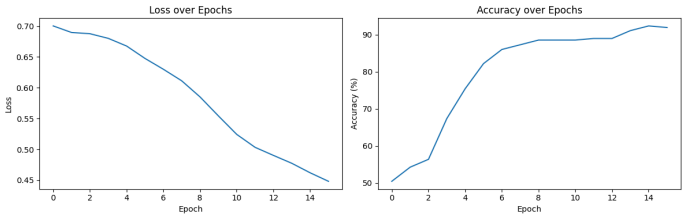
Classical Model:



(5a)

Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 2, Iters: 100, Batch-Size: 256, Rest Args: Default]
Training Accuracy Measurements at 5, 10th, and 15th epoch:
Train: [88.14%, 95.34%, 94.49%]; Test (Final Only): [86.33%]

Quantum Model:



(5b)

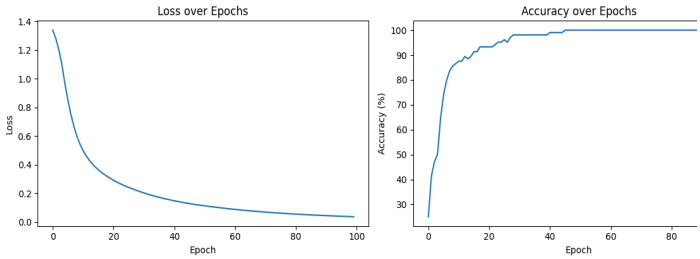
Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 2, Iters: 15, Batch-Size: 256, Rest Args: Default, Learning Rate: 0.1]
Training Accuracy Measurements at 5, 10, and 15th epoch:
Train: [75.42%, 88.56%, 91.95%]; Test (Final Only): [91.41%]

These results indicated extremely close performance to the classical model tested in *Figure [3f]*. Although convergence was slower, the quantum model, in this case, outperformed the classical fair one. In fact, the training and testing accuracies followed much closer in the

quantum setting compared to the classical setting in every comparison, indicating the absence of any significant overfitting.

5.2 4-Class (Multiclass) Classification using ADAM:

Classical Model:



(5c)

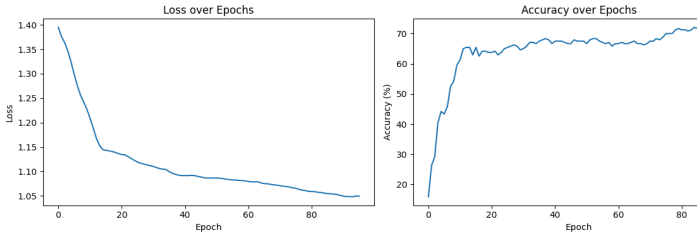
Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 4, Iters: 95, Batch-Size: 256, Rest Args: Default, Learning Rate: 0.1]

Training Accuracy Measurements at 32, 63, and 95th epoch:

Train: [87.92%, 91.67%, 95.00%]; Test (Final Only): [78.13%]

(Earlier checkpoint accuracies also showed signs of overfitting, see Github).

Quantum Model:



(5d)

Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 4, Iters: 95, Batch-Size: 256, Rest Args: Default]

Training Accuracy Measurements at 32, 63, and 95th epoch:

Train: [65.00%, 66.67%, 71.25%]; Test (Final Only): [73.05%]

Unlike the binary class quantum model, which quite rapidly approached the accuracy of its classical counterpart (in a similar number of epochs), the 4-class setup took a much greater number of epochs and converged at an accuracy of more than 5% less than its classical counterpart. Combined with the results from Figure [4a], it is apparent that the function in the quantum setting is more complex to navigate, this is due to the barren plateau problem, or the optimal solution for more complex problems may not be present in the state space. This may indicate a potential problem linked with the complexity of data and circuits increasing the complexity of the function that the optimizer needs to traverse. However, similar to the 2-Class scenario, the testing and training accuracies followed much closer together compared to the classical setting. The training and testing accuracy in the classical setting differed by over 20% (just over 7% at the closest), while the quantum model differed by less than 1.8% and stayed relatively close throughout. This indicates that for gradient-based calculations, at least for Quantum Convolutional Neural Networks (both in binary and multiclass classification models), there might be a potential to avoid overfitting by using a quantum machine learning model, albeit not without tradeoffs in runtime and convergence. This may be useful in applications where the accurate performance of the model is necessary beforehand, which may be required when working with quantum data analysis. A more thorough analysis of overfitting should be performed to reaffirm this analysis.

Nevertheless, for accuracy, it appears that gradient-based algorithms like ADAM are able to significantly outperform gradient-free ones.

Gradient-free calculation using COBYLA, which was found to demonstrate the highest performance among gradient-free optimizers for this problem, demonstrated a significantly lower performance compared to gradient-based calculation (similar to what is expected in classical machine learning). If the pattern of increasing divergence between model accuracies from the 2-class simpler scenario to the 4-class scenario continues, it could indicate that more complicated quantum machine learning architectures with thousands or millions of parameters would require gradient-based optimizers. This could be a significant bottleneck as gradient calculation, even on quantum computers with very high coherence times, scales linearly with parameters: O(n). On the other hand, the final results (~ 60% accuracy on gradient-free and ~ 70% accuracy on gradient-based optimizers for 4-class prediction) arose from very different runtimes (gradient-based run taking 100-500 times the time for one-tenth the epochs), which might indicate that there is

potential to find gradient-free methods to better and more efficiently traverse the Hilbert Space.

6. Increasing Model Accuracy:

Several changes to the architecture were made to test increases in accuracy.

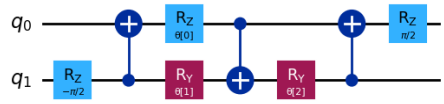
6.1 Altering Model Architecture:

Numerous model architectures were tested with convolutional and pooling layers placed in different arrangements. They did not seem to affect accuracy to a great extent. The results are not included in this paper. Check github implementation for more information.

6.2 Modified Circuit:

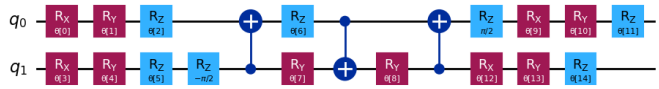
We also test modification to the underlying convolutional circuit. The implementation of the Convolutional Layer was described in Section 2.2. We modify the circuit in Figure [2c] by adding parameterized Rx, Ry, and Rz gates on either side to see the impact.

Original:



[2c],[6a]

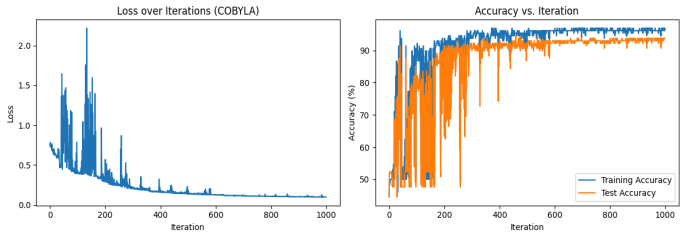
Modified:



[6b]

This approach adds parameters to the circuit and thereby increases circuit complexity significantly. The results are shown below.

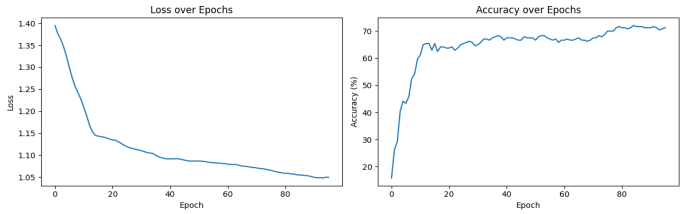
Gradient-Free:



(6c)

Config: [Workflow: PyTorch, Optimizer: COBYLA (Gradient-Free), Loss - CrossEntropyLoss, Classes: 4, Iters: 1000, Batch-Size: 128, Rest Args: Default]
Training Accuracy Measurements at 333, 666, and 1000th epoch:
Best Train: [49.04%]; Test (Final Only): [34.38%]

Gradient-Based:



(6d)

Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 4, Iters: 95, Batch-Size: 256, Rest Args: Default]
Best Train: [75.00%]; Best Test: [73.44%]

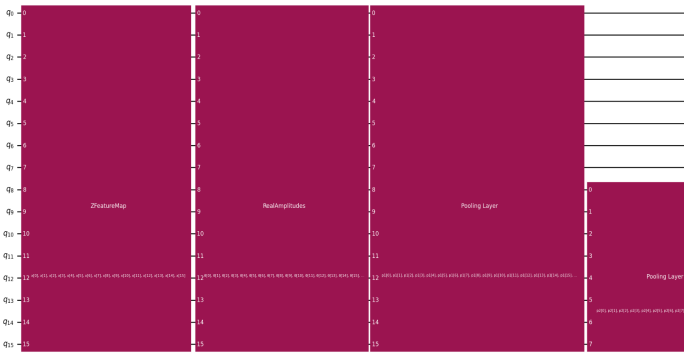
The Gradient-Based optimizer models using the new circuit indicated a marginal improvement from the original convolutional circuit with a train accuracy increase of 3.75% and a test accuracy increase of 0.39%. The gradient-free optimizer models, contrastingly, indicated a sharp decline in train and test accuracy. This falls in line with the pattern seen in classical machine learning of gradient-free methods being unable to find the optimal solution after dimensionality exceeds a couple hundred. This indicates that the problem persists in the Hilbert space, which may be a potential limitation to gradient-free algorithms.

6.3 Classical Processing Layer:

We also tested the effect of classical processing layers to see how much they can skew results and testing of quantum machine learning models.

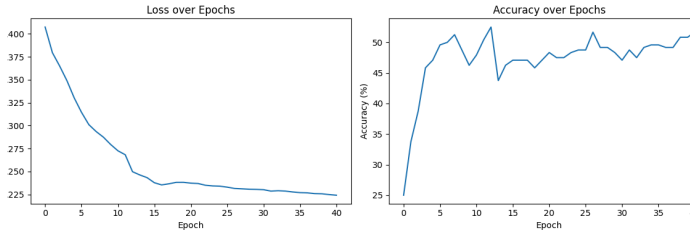
First, we prepare a random circuit with parameters that are entirely quantum. Specifically, we use RealAmplitudes, full entanglement with 2 repetitions. This is followed by

two pooling layers to “encode” the information into 4 qubits for one-hot encoding (see section 2.3).



(6f)

The results are shown below.



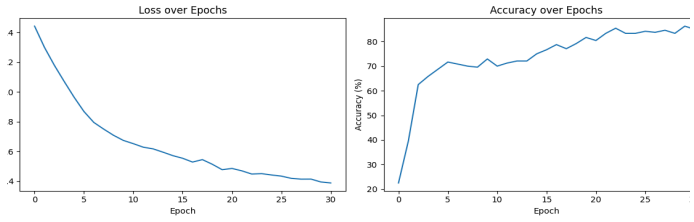
(6g)

Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 4, Iters: 40, Batch-Size: 256, Rest Args: Default]

Best Train: [51.67%]; Best Test: [35.94%]

These results demonstrate the relative effectiveness of the QCNN architecture versus a random quantum circuit. The basic circuit showed significant overfitting and did not manage to get near the 70% accuracy seen for the gradient-based QCNN.

However, when the pooling layer is replaced with a classical linear layer, the accuracy surpasses that of the fully quantum QCNN and even the classical models:



(6h)

Config: [Workflow: PyTorch, Optimizer: ADAM (Gradient), Loss - CrossEntropyLoss, Classes: 4, Iters: 30, Batch-Size: 256, Rest Args: Default]

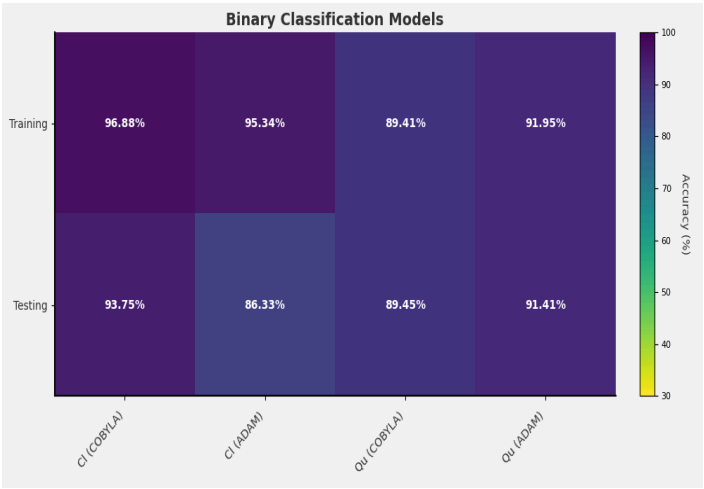
Best Train: [86.25%]; Best Test: [79.69%]

Even the classical model only had a maximum test accuracy of 78.13%, which is lower, albeit marginally, than the testing accuracy of 79.69% achieved by this random circuit combined with a linear layer. The number of parameters in this combined architecture is similar to the number of parameters found in the classical model and the QCNN implementation. Therefore, the hybrid quantum machine learning approach seems to be a promising direction to increase model accuracy.

7. Results Summary

Here we include a final table to summarize the main results. *(‘Qu’ is Quantum, ‘Cl’ is Classical; QRan and QHyb are the circuits used in Figure [6g] and [6h], respectively).*

Heat Map Representation:



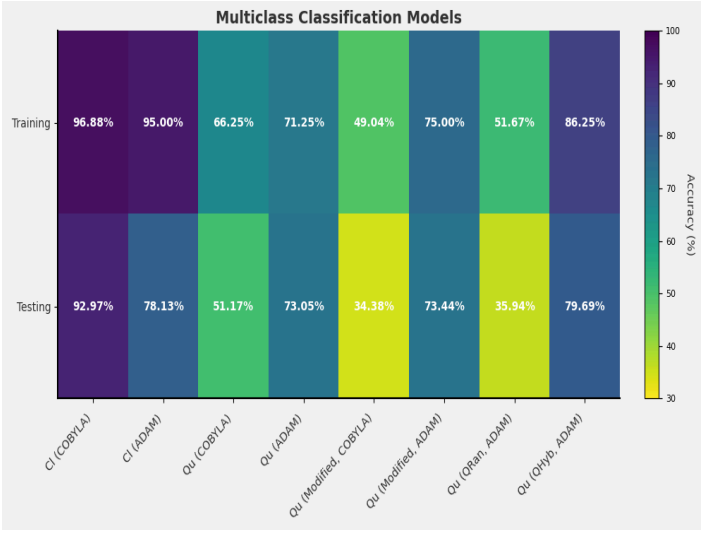
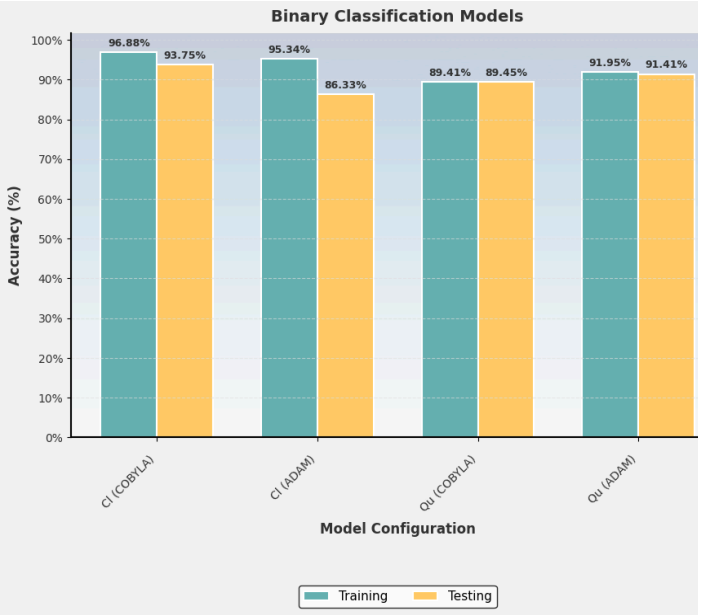
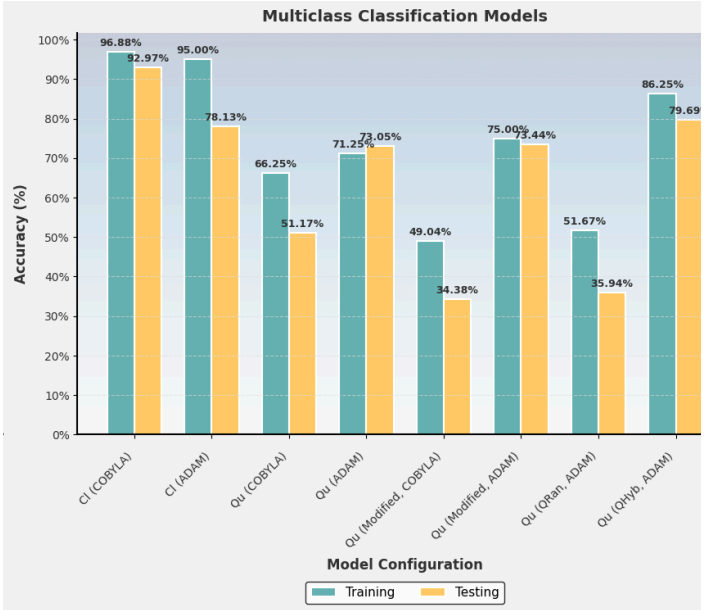


Chart Representation (Same Data):





Classical Cobyla seems to have performed the best in both multiclass and binary classification. It is evident that the classical and quantum ADAM methods both have relatively close training and testing accuracies. The COBYLA method does not perform well in the multiclass scenario, indicating a potential limitation in its ability to traverse more complex functions in quantum machine learning akin to the limitations faced in classical machine learning.

Although our results show that gradient-free optimization does not seem to perform very well for more complex functions, it is an important area to study. Quantum Machine Learning as it is understood at this point may not have many uses for Classical Data, but it may have many uses for quantum data. In that case, the training performed in this paper might be difficult. Firstly, we use EstimatorQNN, which uses shots to perform multiple runs of the circuit. Assuming that quantum data might be difficult to replicate exactly, this approach could require producing the exact same data multiple times, which may be difficult due to the no-cloning theorem. This effect is repeated by the number of epochs, and if you are using gradient-based optimization (like the ParameterShiftRule), by the gradient calculation. If we assume that we would need very deep circuits (in the millions of parameters, such as needed for NLP, etc.), this could become unfeasible. Gradient-Free optimization could allow one to remove part of this bottleneck by eliminating the burden of having to calculate the gradient.

8. Future Prospects

There are numerous directions to expand upon this paper to increase both the efficiency and accuracy of quantum convolutional neural network models.

8.1 Dense Encoding

One potential future direction is to experiment with methods of dense encoding and test whether different encoding methods can have a difference in the prediction of different types of classical data. Specifically, future work could focus on reducing the dimensionality of the input state, allowing information from more pixels to be embedded in the same number of qubits.

8.2 Bottleneck in Gradient-Free Optimizers

Another possible direction would be to more thoroughly understand the difference between Gradient and Gradient-free optimizers on performance by understanding where, if present, there exists a bottleneck for gradient-free optimizers. (Our data suggests that there is likely a bottleneck, but future work could focus on finding exactly where it is and if it differs for different architectures).

8.3 Testing/Developing Optimizers

We mostly looked at the difference between COBYLA and ADAM optimizers. Although we tested numerous optimizers to see which one performed the best, this testing was limited but sufficient to understand the current state of optimizers. It may be useful in the future to examine other gradient-free and gradient-based optimizers to get a deeper understanding of whether the findings presented in this paper hold. There is also a direction to develop and test new optimizers that build on the efficiency of not having to use costly gradients.

8.4 Architectural Changes

Our research focused primarily on the QCNN architecture as it was presented in the original paper and in []. We explored a few limited architectural and circuit-related changes, but future work can involve a more detailed analysis of different architectures building upon this one and more expressive circuits. Additional types of layers can be created and their impact should be analyzed thoroughly. The impact of the number of qubits measured at the end of the quantum circuit on the expressivity of the model should also be measured.

8.4 Noise

One area this paper does not cover is noise. We maintained seeds for our libraries that ensured consistent results and did not replicate the noisy environment of current quantum computers. One could analyze the effect of noise on our findings.

9. Github

Github Link to All Implementations and Testing: https://github.com/hmk55555/QCNN-Research

References

1. Cong, I., Choi, S., & Lukin, M. D. (2018, October 9). *[1810.03787] Quantum Convolutional Neural Networks*. arXiv. Retrieved February 14, 2025, from https://arxiv.org/abs/1810.03787

2. IBM Quantum. (n.d.). *The Quantum Convolution Neural Network - Qiskit Machine Learning 0.8.2*. GitHub Pages. Retrieved February 14, 2025, from https://qiskit-community.github.io/qiskit-machine-learning/tutorials/11_quantum_convolutional_neural_networks.html

3. Vatan, F., & Williams, C. (2024, November 26). *quant-ph/0308006v3 25 Mar 2004*. arXiv. Retrieved February 14, 2025, from https://arxiv.org/pdf/quant-ph/0308006